



Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Doctoral Thesis
Supervised by
Benoit Baudry and Martin Monperrus
Stockholm, Sweden, 2023

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2020:4
ISBN 100-

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges
till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik
i .

© Javier Cabrera-Arteaga , date

Tryck: Universitetsservice US AB

Abstract

[1]

Keywords: Lorem, Ipsum, Dolor, Sit, Amet

Sammanfattning

[1]

LIST OF PAPERS

1. ***Superoptimization of WebAssembly Bytecode***
Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs
<https://doi.org/10.1145/3397537.3397567>
2. ***CROW: Code Diversification for WebAssembly***
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Network and Distributed System Security Symposium (NDSS 2021), MADWeb
<https://doi.org/10.14722/madweb.2021.23004>
3. ***Multi-Variant Execution at the Edge***
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Conference on Computer and Communications Security (CCS 2022), Moving Target Defense (MTD)
<https://dl.acm.org/doi/abs/10.1145/3560828.3564007>
4. ***WebAssembly Diversification for Malware Evasion***
Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023
<https://www.sciencedirect.com/science/article/pii/S0167404823002067>
5. ***Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly***
Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
6. ***Scalable Comparison of JavaScript V8 Bytecode Traces***
Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (SPLASH 2019)
<https://doi.org/10.1145/3358504.3361228>

ACKNOWLEDGEMENT

[1]

ACRONYMS

List of commonly used acronyms:

AE Acronym examples

Contents

List of Papers	iii
Acknowledgement	iv
Acronyms	v
Contents	1
1 Introduction	2
1.1 Background	2
1.2 Problem statement	2
1.3 Automatic Software diversification requirements	2
1.4 List of contributions	2
1.5 Summary of research papers	3
1.6 Thesis outline	3
2 Background and state of the art	4
2.1 WebAssembly	4
WebAssembly toolchains	4
2.2 Software diversification	4
2.3 Generating Software Diversification	4
Variants generation	4
Variants equivalence	4
2.4 Exploiting Software Diversification	4
Defensive Diversification	4
Offensive Diversification	4

3	Automatic Software Diversification for WebAssembly	5
3.1	Approach landscape	5
3.2	Compiler based approaches	5
	CROW: Code Randomization of WebAssembly	5
	Exploration.	6
	Constant inferring	8
	Removing subsequent optimizations for LLVM	9
	MEWE	9
	Multivariant binaries	9
3.3	Binary based approach	9
4	Evaluation	10
4.1	Use cases	10
4.2	Experimental protocols	10
	Metrics	10
4.3	Results	10
5	Results and discussion	11
5.1	Summary of technical contributions	11
5.2	Summary of empirical findings	11
5.3	Summary of empirical findings	11
5.4	Future Work	11
I	Included papers	12
	Superoptimization of WebAssembly Bytecode	14
	CROW: Code Diversification for WebAssembly	15
	Multi-Variant Execution at the Edge	16
	WebAssembly Diversification for Malware Evasion	17
	Wasm-mutate: Fast and Effective Binary Diversification for WebAssembly	18

CONTENTS

3

Scalable Comparison of JavaScript V8 Bytecode Traces

19

TODO Recent papers first. Mention Workshops instead in conference. "Proceedings of XXXX". Add the pages in the papers list.

■ 1.1 Background

TODO Motivate with the open challenges.

■ 1.2 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

■ 1.3 Automatic Software diversification requirements

1. 1: **TODO** Requirement 1

■ 1.4 List of contributions

C1: Methodology contribution: We propose a methodology for generating software diversification for WebAssembly and the assessment of the generated diversity.

C2: Theoretical contribution: We propose theoretical foundation in order to improve Software Diversification for WebAssembly.

C3: Automatic diversity generation for WebAssembly: We generate WebAssembly program variants.

C4: Software Diversity for Defensive Purposes: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5: Software Diversity for Offensives Purposes: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

Contribution	Resarch papers				
	P1	P2	P3	P4	P5
C1	x	x		x	x
C2	x	x			
C3	x	x	x		
C4	x	x	x		
C5			x		
C6	x	x	x	x	x

Table 1.1: Mapping of the contributions to the research papers appended to this thesis.

C6: Software Artifacts: We provide software artifacts for the research community to reproduce our results.

TODO Make multi column table

■ 1.5 Summary of research papers

- Paper 1:** Superoptimization of WebAssembly Bytecode.
- Paper 2:** CROW: Code randomization for WebAssembly bytecode.
- Paper 3:** Multivariant execution at the Edge.
- Paper 4:** Wasm-mutate: Fast and efficient software diversification for WebAssembly.
- Paper 5:** WebAssembly Diversification for Malware evasion.

■ 1.6 Thesis outline

02

BACKGROUND AND STATE OF THE ART

■ 2.1 WebAssembly

■ 2.1.0 WebAssembly toolchains

TODO Mention, stress the landscape of tools that involve Wasm. Include analysis tools, fuzzers, optimizers and malware detectors.

TODO End up motivating the need of Software Diversification for: testing and reliability.

■ 2.2 Software diversification

■ 2.3 Generating Software Diversification

■ 2.3.0 Variants generation

■ 2.3.0 Variants equivalence

■ 2.4 Exploiting Software Diversification

■ 2.4.0 Defensive Diversification

■ 2.4.0 Offensive Diversification

git config pull.rebase false **TODO** Start here. 4 pages each and 2 pages discussion. Target 20 pages.

■ 3.1 Approach landscape

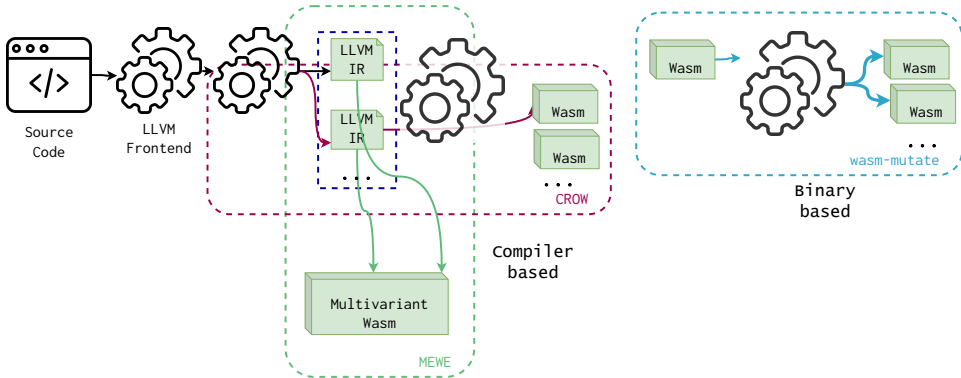


Figure 3.1: Approach landscape.

■ 3.2 Compiler based approaches

■ 3.2.0 CROW: Code Randomization of WebAssembly

This section describes the red squared tooling in ?? named CROW [?]. CROW is a tool tailored to create semantically equivalent Wasm variants from an LLVM front-end output. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

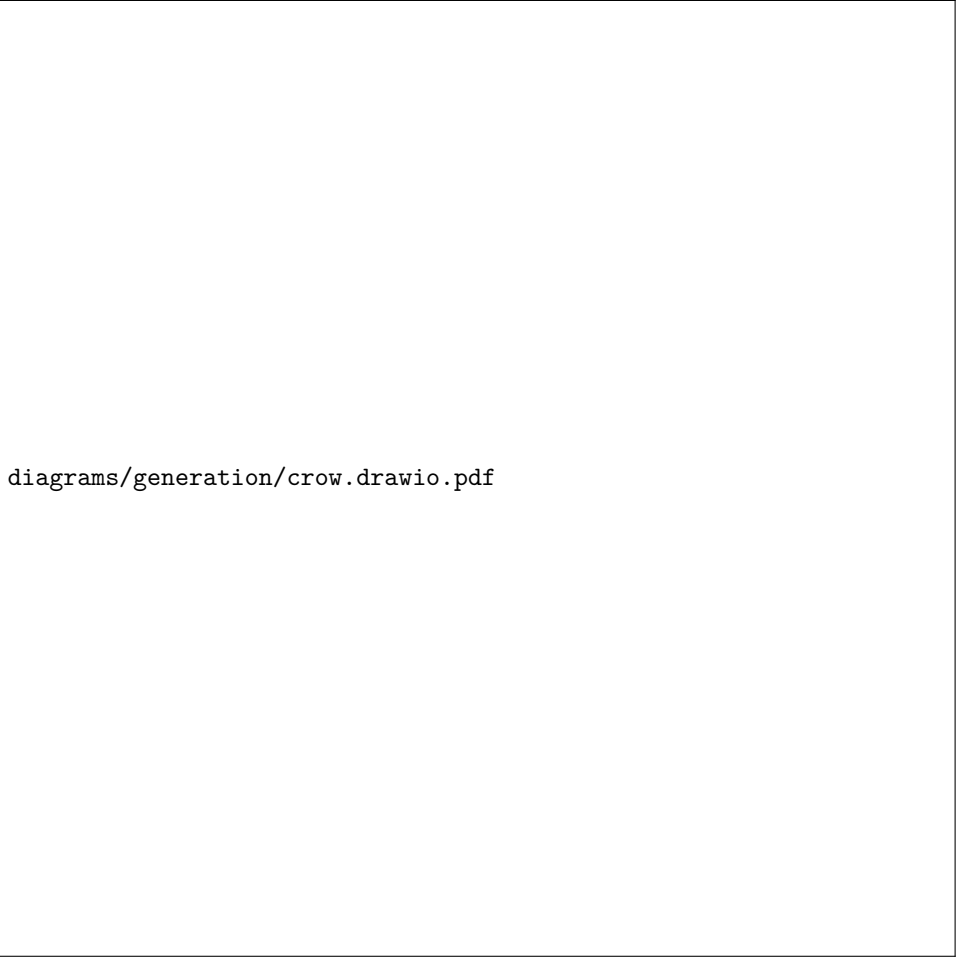
In Figure 3.2, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function in its input LLVM. For all LLVM instructions, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into Wasm binaries. In the following text, we describe our design decisions. All our implementation choices are based on one premise: *each design decision should increase the number of Wasm variants that CROW creates.*

■ 3.2.0 Exploration

The primary component of CROW’s exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the proposed superdiversifier of Jacob et al. [?]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [?] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine’s instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs is found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the superdiversifier idea of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on handwritten rules. Concretely, we can control the quality of the generated codes. Besides, CROW always generates equivalent programs because it is based on a solver to check for equivalence. Second, there is a battle-tested superoptimizer for LLVM, Souper [?]. This latter makes it feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation is the optimal replacement result of the searching. CROW keeps more equivalent replacements during the searching by removing the halting criteria. Instead the original halting conditions, CROW does not halt when it finds the first replacement. CROW continues the search until



diagrams/generation/crow.drawio.pdf

Figure 3.2: CROW components following the diagram in ???. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

a timeout is reached or the replacements grow to a size larger than a predefined threshold.

Notice that the searching space increases exponentially with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the Wasm backend. This decision reduces the searching space. For example, creating an expression having the **freeze** LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning

strategies of Souper for the sake of more program variants. For example, Souper prevents the generation of the commutative operations during the searching. On the contrary, CROW still uses such transformation as a strategy to generate program variants.

■ 3.2.0 Constant inferring

One of the code transformation strategies of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in ??.

After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```

1  int babbage() {
2      int current = 0,
3      square;
4      while ((square=current*current) %
5             ↪ 1000000 != 269696) {
6          current++;
7      }
8      printf ("The number is %d\n", current)
9             ↪ ;
10     return 0 ;
11 }
```

In theory, this value can also be inferred

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```

int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current);
    return 0 ;
}
```

by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **while**-loop because the loop count is too large. The original Souper deals with this case, generating the

program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime¹.

■ 3.2.0 Removing subsequent optimizations for LLVM

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the Wasm backend. We disable all optimizations in the Wasm backend that could reverse the CROW transformations. In the following enumeration, we list three concrete optimizations that we remove from the Wasm backend.²

- Constant folding: this optimization calculates the operation over two (or more) constants in compiling time, and replaces the original expression by its constant result. For example, let us suppose $a = 10 + 12$ a subexpression to be compiled, with the original optimization, the Wasm backend replaces it by $a = 22$.
- Expressions normalization: in this case, the comparison operations are normalized to its complementary operation, e.g. $a > b$ is always replaced by $b \leq a$.
- Redundant operation removal: expressions such as the multiplication of variables by $a = b2^n$ are replaced by shift left operations $a = b \ll n$.

■ 3.2.0 MEWE

■ 3.2.0 Multivariant binaries

■ 3.3 Binary based approach

¹Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in ??.

²We only illustrate three of the removed optimization for the sake of simplicity.

■ 4.1 Use cases

RQ1: Defensive Diversification: ?

RQ2: Offensive Diversification: ?

■ 4.2 Experimental protocols

■ 4.2.0 Metrics

New static metric. Diversification preservation.

■ 4.3 Results

- 5.1 Summary of technical contributions
- 5.2 Summary of empirical findings
- 5.3 Summary of empirical findings
- 5.4 Future Work

REFERENCES

Part I

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus

Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (Programming 2021), MoreVMs

<https://doi.org/10.1145/3397537.3397567>

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry,
Martin Monperrus

Network and Distributed System Security Symposium (NDSS 2021), MADWeb

<https://doi.org/10.14722/madweb.2021.23004>

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
*Conference on Computer and Communications Security (CCS 2022), Moving
Target Defense (MTD)*

<https://dl.acm.org/doi/abs/10.1145/3560828.3564007>

WEBASSEMBLY DIVERSIFICATION FOR MALWARE EVASION

Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023

<https://www.sciencedirect.com/science/article/pii/S0167404823002067>

WASM-MUTATE: FAST AND EFFECTIVE BINARY DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Nick Fitzgerald, Martin Monperrus, Benoit Baudry
Under revision

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry

*11th ACM SIGPLAN International Workshop on Virtual Machines and
Intermediate Languages (SPLASH 2019)*

<https://doi.org/10.1145/3358504.3361228>